Converting UML Models From Telelogic TAU To XMI

Sami Liedes

12th April 2007

Abstract

To combat bugs in complex software systems, symbolic methods for UML model verification are being developed in the SMUML project. In this work, a converter from the file format of an industrially used UML tool, the Telelogic TAU application, to the XMI UML 1.4 format used in the SMUML project is designed. The goal is to be able to handle a significant amount of the TAU language, mainly related to behavioral modeling using state machines.

The converter uses a standard Python XML toolkit to parse the TAU "u2" file format, and the open source metamodeling toolkit Coral to produce the XMI output. UML 1.4 is generated instead of UML 2.0 because a Coral UML 2.0 metamodel does not yet exist. This poses some challenges, because the version of UML used in TAU is in some aspects closer to UML 2.0. Translation is divided into two phases, one program that converts TAU models into Coral models which conform to a TAU metamodel implemented in Coral ("Coral TAU" models), and one that converts these Coral TAU models further into Coral models which conform to the UML 1.4 model shipped with Coral ("Coral UML 1.4" models). Both of these phases are based on traversing the model tree and mapping the concepts in the tree to corresponding target concepts. The first phase, the conversion of simple state machine models from TAU models into Coral TAU models, is implemented. A detailed design for the second phase is presented.

As UML does not define an action language for the state machine diagrams, an action language named Jumbala is used. Conversion from a restricted subset of the TAU action language to Jumbala is outlined.

Contents

1 Introduction

Present-day software systems are increasingly complex feats of engineering. As such they are inevitably bound to have bugs in them. This problem is magnified by the requirements of the business world as the systems need to be developed and deployed as fast as possible, often leaving little time for proper design and testing.

In addition to the software that runs on a PC, there are many appliances that have hidden software in them. In the modern world even washing machines usually contain software.

Often nowadays software also plays a role in applications where failure cannot be tolerated because it could mean loss of human life or a severe financial setback. Modern cars contain complex programs to control what happens when you press your gas pedal. Medical devices contain code to control the amount of x-rays the patient gets, or to control the patient's heart beat. Space stations, rockets and satellites are just too expensive to be lost because of a software failure.

There are many ways to combat bugs in software, with careful design and rigorous testing being the traditional solutions. However, applying formal methods to critical pieces of software has lately been gaining popularity. Formal methods have the advantage over testing that they can be used to prove the lack of certain kinds of bugs in software as opposed to mere relative confidence provided by traditional testing. Also applying formal methods to real world programming tasks is a relatively new art; we envision a future where formal methods will be a cost-effective approach to ensuring bug-free software [7].

The SMUML, or Symbolic Methods for UML Behavioural Diagrams, project[1] aims to use formal methods to verify behavioral properties of systems modeled in UML [9]. For reading UML models, the front end in the SMUML toolchain uses the Coral [1] metamodeling toolkit, which is able to handle UML 1.4 models. Industrially, however, other tools are used. One of them is the commercial TAU application by Telelogic. TAU [12] is a UML modeling application used by the industrial partners of the SMUML project. Hence, conversion needs to be done to support models provided by the industrial partners.

In this work an automatic converter from TAU models to Coral models is designed. See Figure 1 for how this fits in the big picture. The converter needs to support the model elements used in the industry, or a subset of them as the aim of the SMUML project is to develop proof-of-concept code, not polished end products. So far this design can handle only simple models. Support for more advanced features is left for future work.

[1] A project of the Laboratory for Theoretical Computer Science at Helsinki University of Technology

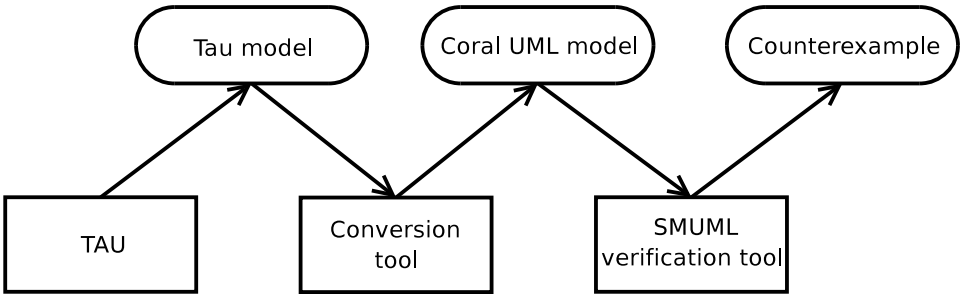

Figure 1: The TAU to Coral conversion in the context of the SMUML toolchain.

The UML models being verified contain active classes, whose behavior is modeled using state machines. As the action language, the Jumbala [4] language is used. The action language is used in the transitions of the state machines to specify actions to be done when moving from one state to another.

Complex structures of the action language and advanced structures of the UML language are not yet supported.

2 Central concepts

2.1 Unified Modeling Language

Unified Modeling Language, or UML, is the de facto industrial standard modeling language with which many different aspects of the system being designed, its requirements and functionality can be captured in design documents [6]. UML provides the tools to model the system in any level considered appropriate, from very high level to low-level designs. When models are presented in UML, specific subsets of the language allow for simulating and testing both high-level and low-level designs and testing the behavior of some aspects of the model or generating code for the final product.

UML defines a graphical notation for expressing designs, with a number of different kinds of diagrams. Perhaps one of the most basic kinds of diagrams is the class diagram. The class diagram captures the relationships of classes. In a simple system this can possibly mean all classes, while in a more complex system the classes presented in a single diagram would be the central classes of the application or those central to a subsystem being documented. A class diagram shows classes, their attributes (variables owned by the classes) and operations (methods).

Additionally relationships between the classes are shown. These may be mere association (i.e. the classes have some collaboration), aggregation (one of the classes contains the other) or composition (one of the classes owns the

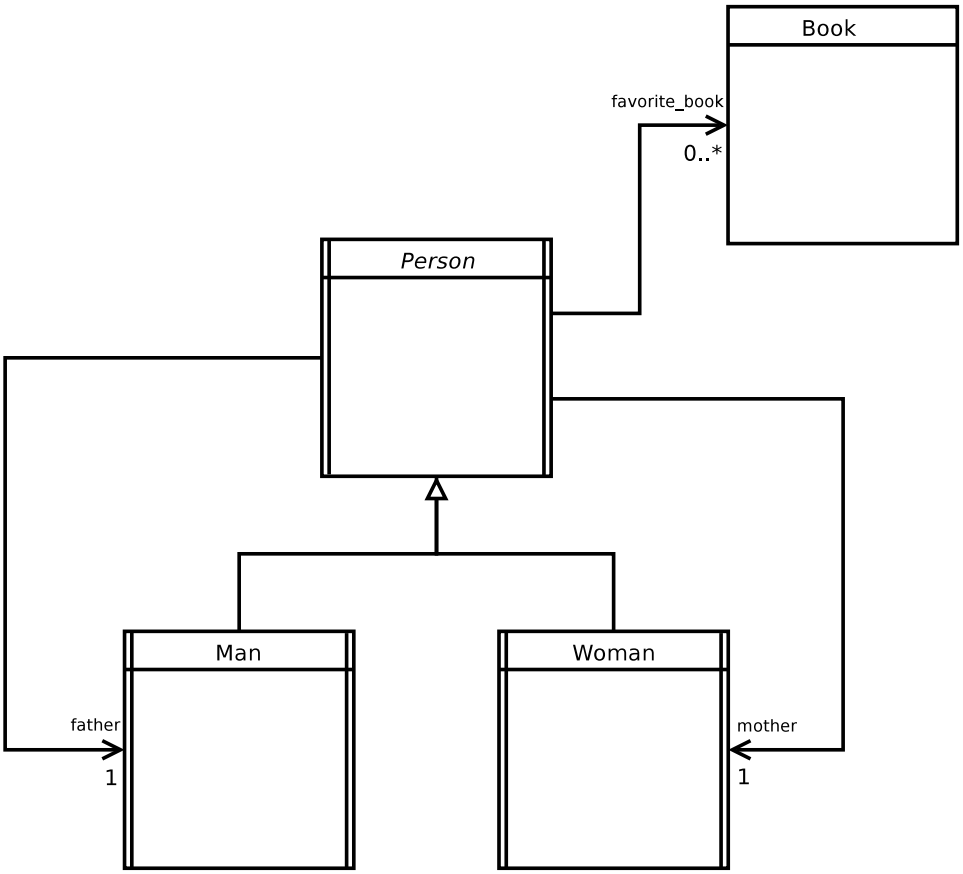

Figure 2: An example of a class diagram. In this diagram there is an abstract active class named *Person* with two subclasses, *Man* and *Woman*. Every *Person* has a *father* and a *mother* and zero or more favorite books.

other). In addition there can be a generalization relationship, meaning that one of the classes is a subclass of another.

A class can be *active*, indicating that the class can execute actions spontaneously. In practice this means that the class either has its own thread of execution or actually represents some external object or the user. Active objects communicate asynchronously by sending signals to each other. An active class has its side bars doubled in a class diagram. See Figure 2 for an example of a class diagram.

Another kind of diagram in UML is a state machine diagram [6]. It can be used to describe the behavior of an instance of an active class. See Figure 3 for an example of a state machine diagram.

A state machine diagram describes a state machine. UML state machines are based on finite state machines, a fundamental concept in computer science. A system described by a state machine is assumed to be in a single

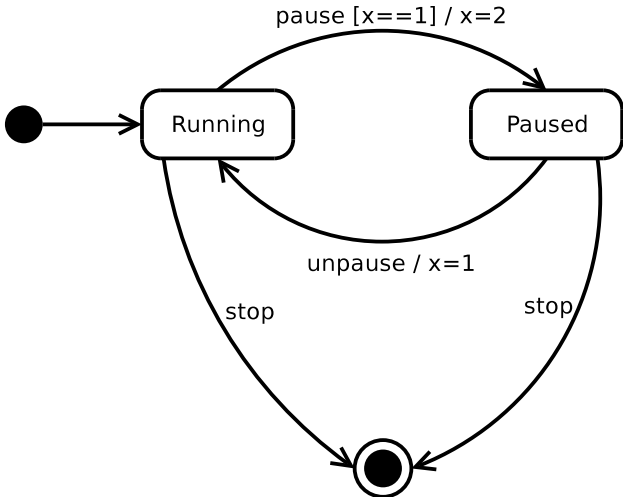

Figure 3: An example of a state machine diagram, modeling a player of some kind.

state at any point in time. The diagram captures the high level structure of the operation being modeled in an easily understandable way. A state machine further specifies how the state of the system can change in a controlled way during execution.

There are a few different kinds of items in a UML state machine diagram. A filled black circle is an *initial state*. The execution of the state machine starts from it. There are the normal states, represented by rectangles with rounded corners. A *final state* is represented by a filled black circle inside a bigger hollow circle. The execution of the machine ends when it reaches the final state.

Between states there are *transitions*, which are represented as arrows from a state to another. Transitions are the way in which the state of the machine can change, for example the state of a player from a playing state to a paused or stopped state as in Figure 3.

Transitions generally have *triggers*, *guards* and *actions*. Triggers specify which signal the class being modeled by the state machine must receive in order to transition from the source state to the target state. A guard places a condition on the transition; if the guard does not evaluate to true, a transition cannot be fired. In Figure 3 a transition from the *Running* state to the *Paused* state requires, in addition to receiving a *pause* signal, that the value of the variable x is 1. A guard must not have side effects. An action specifies program code to be run when the transition is fired, just after leaving the source state and before entering the target state. In Figure 3, in the above-mentioned transition, the value of x is set to 2.

Sometimes it is useful to model internal behavior inside a single state. For

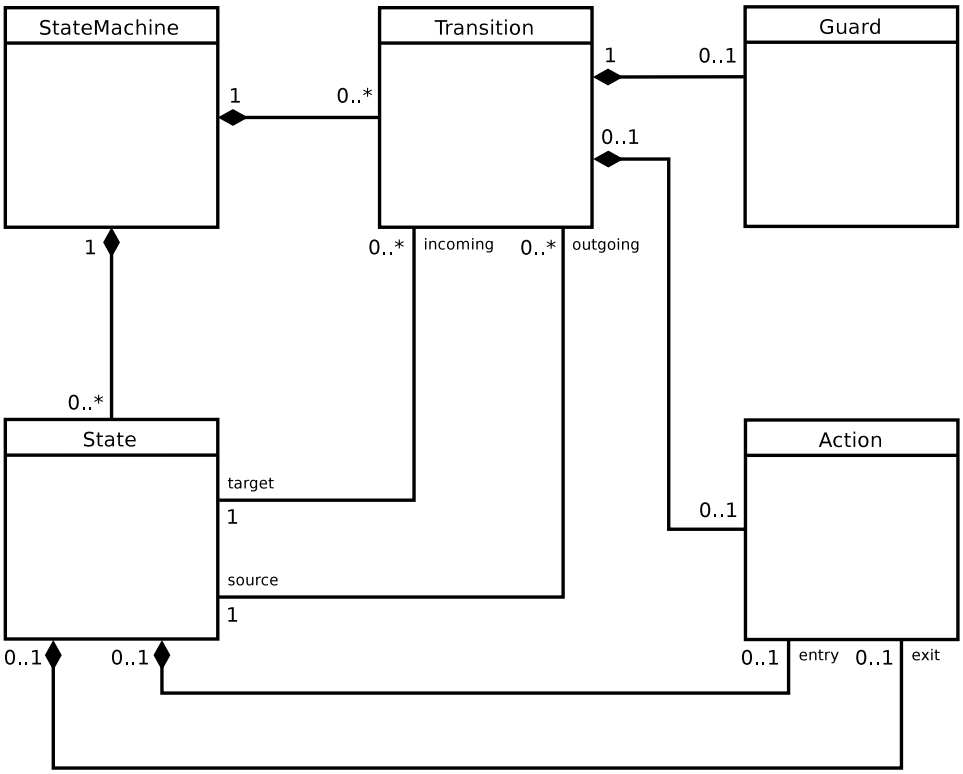

Figure 4: An example of a metamodel, describing state machines.

such cases UML provides *composite states*. A composite state is essentially a state machine inside a state. They can be used to model the behavior of a complex system with a state machine diagram in a more structured way.

See [2], pp. 297-318, for a more thorough description of state machines and their role in modeling software systems.

2.2 Metamodels

A metamodel is a model that describes models [6]. It does this by representing all the concepts of the models being metamodeled and their relationships. For example, in a metamodel describing UML state machines there would be a metaclass named *State*, one named *Transition* and so on. See Figure 4, a simplified metamodel for models of state machines adapted from one presented in [6].

In Figure 3, for example, the states *Running* and *Paused* are instances of the metaclass *State* in Figure 4. Similarly, $x == 1$ is an instance of the metaclass *Guard*, while $x = 2$ and $x = 1$ are instances of the metaclass *Action*.

2.3 The SMUML project

The aim of the SMUML, or Symbolic Methods for UML Behavioural Diagrams, project is to develop tools for analyzing state machine diagrams in UML models using symbolic verification methods [9]. State machines are extracted from the model and corresponding input for a stock verification tool is generated. A counterexample generated by the model verifier can be translated back to the UML realm and examined using the SMUML UML simulator.

UML is action language agnostic. It does not specify the action language in any way; it merely treats pieces of action language as strings. For verification purposes it is imperative that an action language be specified. Jumbala [4] is the action language used in the SMUML project. Jumbala is mostly simplified Java extended with the ability to send and receive signals. While Jumbala is simplified Java, the core features of Java, most importantly object management, have been preserved in it. Jumbala can also be used to traverse association instances, called links, between classes in the UML model.

2.4 Coral

Internally the tools developed in the SMUML project use the XMI format as supported by the open source metamodeling tool Coral [1]. Coral is used in reading models from files and in handling them. It is a toolkit that can be used to create, edit and transform UML models as well as models conforming to other metamodels. This can be done using the Python language. It is also possible to develop new metamodels for Coral.

3 Problem statement

It would be desirable for it to be possible to use existing industrial tools in addition to Coral to design the UML models subsequently verified using symbolic methods. One such industrial tool is the Telelogic TAU application.[12]

The conversion of models designed in TAU to XMI format is not entirely straightforward for a number of reasons. First of all TAU is not entirely a UML program; it shows signs of not having been originally designed for UML, but SDL, another modeling language used especially in the telecommunications industry. Often models designed in a way natural to TAU, although mostly UML, have SDLisms in them.

Secondly, TAU models mostly conform to UML 2.0, while Coral currently only has a UML 1.4 metamodel (a UML 2.0 metamodel is being developed). When converting TAU models to Coral, some logic is needed to either convert UML 2.0 features to corresponding UML 1.4 ones, or to strip the model of

those features that are not supported while still retaining those properties of the model that are desired for verification purposes.

Thirdly, TAU has its own action language for state machine actions, while Coral does not commit to any action language. In the SMUML project Jumbala is used. Therefore it is necessary to handle the action language of TAU at least to some extent. [3]

Fourthly, exporting models from TAU is not very easy. While there is a C++ API for inspecting the model, exporting the model in any format not native to TAU will require traversing the model tree and generating the corresponding target model (as opposed to e.g. XMI export, after which it could be possible to employ rather simple rewriting rules). Parsing the proprietary file format of TAU, called "u2", is also slightly problematic as the format is undocumented. However at least in current versions of TAU the format is in XML format with rather human-readable tag names, so this is not as big a problem as it could be.

Some of the simpler features that the converter needs to handle are:

- packages
- classes
- signals (to some extent)
- signal parameters
- attributes
- state machines
- composite states
- decision actions
- some TAU action language
- associations
- aggregations
- compositions

These are considered in detail in the following sections.
Some of the more complex features are:

- ports (UML 2.0)
- most of the TAU action language

These are mostly left for future work.

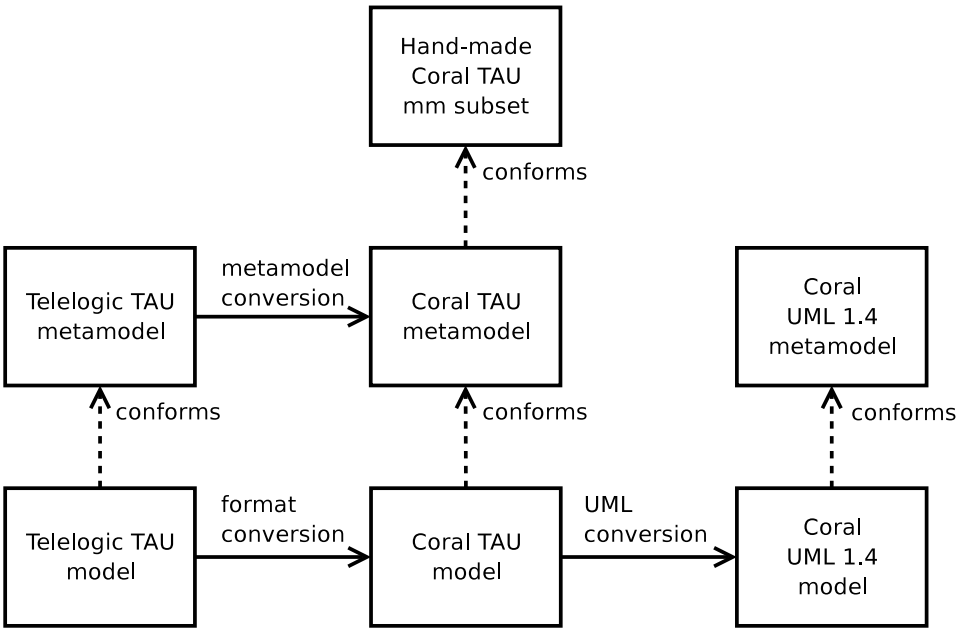

Figure 5: Overview of the different kinds of conversions.

4 Design of the conversion

In order to tackle the complexity, the task of converting TAU models into Coral UML 1.4 models was split in two parts which can be implemented in separate programs. The first program converts TAU models into Coral models that conform to an automatically generated Coral TAU metamodel. The second program takes Coral models that conform to the Coral TAU metamodel and converts them to Coral UML 1.4 models. These conversions are illustrated in Figure 5.

4.1 Metamodel conversion

The Telelogic TAU metamodel is a TAU model provided in "u2" format in the TAU SDK. It describes the structure of TAU models. The Coral TAU metamodel is a metamodel understood by the Coral toolkit. It too describes TAU models.

The first subset of the Coral TAU metamodel, done by hand, had to include elements that occur in the TAU metamodel. These include packages, classes, attributes, generalizations, associations (including aggregations and compositions), data types, ranges and literals, as well as some peculiarities of the TAU tool, such as dependencies.

To make the conversion between Telelogic TAU models and Coral TAU models as simple and reliable as possible, an automatic conversion from the

Telelogic TAU metamodel to a corresponding Coral metamodel was done. The conversion uses as its source the TAU model of the metamodel, provided in the TAU SDK. The conversion reads the metamodel model and converts it to a Coral model that conforms to the metamodel done by hand. After this the Coral TAU metamodel model (i.e. a Coral model, in XMI format) is further converted to a format understood by a Coral metamodel compiler script, a simple line-based "nat" format, and further compiled into the Coral's own XML-based Simple Metamodel Description (SMD) format [11], a subset of UML Meta Object Facility (MOF) format. Coral uses SMD for metamodels.

The script that implements this conversion needs to be able to read the constructions implemented in the hand-made Coral TAU metamodel subset and to write a Coral model conforming to it. The Coral model can be generated using the Python API of the Coral tool, which is implemented in Python and C++, and saved using a Coral routine. Hence the tool only needs to know about the u2 format and not about the file format used by Coral.

This conversion is not entirely trivial: there is the question of what to do with data types. Currently our implementation converts nearly all references to "primitive" data types into string attributes in the corresponding Coral model conforming to the Coral TAU metamodel. This means that if the type of a field in the TAU model is Integer, the corresponding field in the converted output would be a string with a note that the contents of that field is an integer.

4.2 Telelogic TAU model to Coral TAU model conversion

There are two approaches to parsing TAU models. The first possibility is to traverse the model by using the API of TAU, or actually a Python wrapper over the C API kindly provided by Markku Turunen from Nokia. Traversing the model using the TAU API is relatively simple, however unfortunately there have been some problems to this approach. We could not find out if it was possible to determine whether a link traversed is an association or a composition. Furthermore, the Python wrapper does not work with TAU 2.7 even with some trivial modifications to make it compile, but instead crashes with "Aborted" when loading a project.

The second approach is the parsing of the "u2" format files created by TAU. The u2 format is XML based and fairly straightforward. Of course there is some work to do in this approach that could be avoided when traversing the model with an API provided by TAU. Mostly this means having to resolve references between the elements in the model – each element is assigned a globally unique identifier (GUID) – and having to support loading multiple u2 files as the model can span several files instead of simply loading a single project file.

A drawback in parsing the u2 file format is that it is possible for it to change substantially between TAU versions. In any case a u2 parser written for a simplified subset of the TAU 2.5 metamodel worked with no modifications at all with TAU 2.7. At a detailed level, the metamodels of TAU versions 2.5 and 2.7 do have some superficial differences.

4.3 Coral TAU model to Coral UML 1.4 model conversion

A program is needed for converting Coral TAU models into Coral UML 1.4 models. This is the most complex part of the whole conversion. While we believe it is reasonable easy to support even very complex TAU models in the first phase of the conversion, in this phase relatively big effort is needed for each additional feature.

The Coral TAU model can be easily read using the Coral Python API. Other than that, this part of the chain is in principle similar to the first one, a matter of doing a mapping operation to a tree. The details are much more complex, however, because of the conversions we want to do to the model. Also mapping the action language needs to be considered, which is a rather separate and nontrivial task. In this work we propose mappings for some of the simpler constructs of the TAU models and the TAU action language. Support for the more complex features of the action language is left for future work.

One problem when converting TAU models into Coral models is what to include and what to leave out. TAU comes with an extensive predefined library of data types, operators and the like. It is a nontrivial design decision to draw a line between elements to be converted and those that are left out. The current approach relies on a minimal dependency on the TAU predefined elements to keep the Coral Tau model simple, but this may need to be changed to be able to convert more complex models like those modeled using the Lyra method as described in [8].

When traversing the tree and mapping constructs to equivalent UML 1.4 ones, it is possible for a reference to some element in the model tree to appear before that element has been processed. Something needs to be done to handle that case. Hence some forward references in the model tree need to be spared until all the elements themselves have been translated. This means that the conversion has to be done in two passes.

To handle these forward references that cannot be properly resolved in one pass, a reference mapper object can be created. In object oriented programming terms it is a singleton and acts as a kind of cache for the object conversion. It takes a Coral TAU model object and a Python class as input. If the object is one that it has never before seen, it returns a new instance (instantiated with the default constructor) of the passed class. Otherwise it returns the object created on the first time. In this way it prevents the mapping of a single TAU object to several different Coral objects, instead

returning the Coral object created when the corresponding TAU object was first encountered. Thus the purpose of this "cache" is not to speed up anything but to ensure a 1:1 mapping.

The problems in this case are related to the differences between the TAU metamodel and UML 1.4. TAU models can contain some objects, for example ports, that were only added to UML 2.0 and so are missing from UML 1.4. In this work, we have not put much thought into how the conversion of ports should be done, but it can be expected to require to a nontrivial amount of work.

Some things are simple to convert. These include packages, classes, signals, attributes and data types. Also the conversion of expressions is a relatively simple matter of tree-mapping. However the conversions of several different kinds of Coral TAU model objects are quite complex. Examples of complex conversions are those of transitions and state machines, because nested state machines are converted to composite states.

In the following we detail the conversion for the different types of objects that need to be handled.

4.3.1 Packages

Packages are easy to convert. TAU packages are very similar to UML 1.4 ones. They share with classes the property that they can contain other packages or classes, hence some common code can be used for them. In TAU the contained objects, whether classes, packages, signals or something else, are all contained in a single attribute named "ownedMember".

When a Coral TAU package is read, a UML 1.4 package is created and all the contained objects of the TAU package are mapped to corresponding UML 1.4 packages (recursively) and added as children to the UML 1.4 package. The only converted primitive attribute for a package is its name.

4.3.2 Classes

Classes are quite similar to packages. In addition to having a name, a class also has a property that tells if it is active or not.

Another difference to packages is that classes are types while packages are not. This is taken into account by adding each class processed to a type dictionary where they are readily available when a non-primitive type is encountered in some context.

4.3.3 Signals and ports

Signal mapping is not as trivial as the conversions described above. The same signal can be sent in multiple different places, and this needs to be determined from the signal name. In UML 1.4, where the signal occurs is stored in the signal itself in a field named *occurrence*, so that information

needs to be gathered. The parameters need to be converted, too. Coral supports (ordered) lists as attributes, so the order can be preserved without extra work. The parameter types also need to be resolved by a lookup from the type dictionary.

Functionality equivalent to TAU/UML 2.0 ports is challenging to implement in UML 1.4. When sending the signal, we effectively need to know the object to which we are sending it. The details of this conversion is left for future work.

4.3.4 Attributes

Classes can have attributes. In TAU, also packages can have attributes. Package attributes can be converted to static member variables in a class; a special class can be made for each package to contain the attributes. In addition to this, visibility needs to be taken into account. Attributes can be either public, protected or private as is standard in object oriented design. This information is simply mapped from the Coral TAU object format to the directly corresponding Coral UML 1.4 object format.

4.3.5 Data types

Data types can be encountered in multiple places, typically inside packages and classes. The simple implementation of this will not take definition scopes into account, so it assumes there are no two types with the same name even in different scopes. A complete implementation of scopes would take into account the limited visibility of each data type. However, this would be rather complex compared to the simple way of doing this, and we argue that while it would have to be implemented for a productized converter, it would not be worth the benefits for this research work.

Primitive data types like integers, booleans and strings are easy. Indeed Coral and UML 1.4 + Jumbala support them as built-in types, so the only conversion needed has been done in the first phase of the conversion and in this phase they can be copied intact from the Coral TAU model. Types that are references to other classes are likewise easy; they are essentially similar in TAU and UML 1.4 + Jumbala. We are unsure if TAU has some C-like struct type separate from classes, but it it does, it can easily be converted to a class with public members.

TAU has one special built-in type, the "pid" type, that poses some challenges. It can be modeled in Jumbala by wrapping it in a class. One way of implementing this is making a common superclass (or actually in Java terms an interface) for all active classes. After this, the pid type can be implemented as a reference to instances of this class. However it must be noted that the SMUML toolchain doesn't at least currently support inheritance.

4.3.6 Associations, aggregations and compositions

Associations, aggregations and compositions are a bit more tricky thanks to how they are represented in TAU. In TAU the endpoints of the associations are indistinguishable from attributes in the involved classes. We believe the easiest solution to this is to first actually translate the attributes as if they were not associations, since it is impossible to know at that point in conversion whether they are real attributes or not. When converting associations, the targets and their respective attributes can be added to a global list and later deleted from the resulting UML 1.4 model.

Also sometimes when parsing the TAU model the association objects can be encountered earlier than the related classes, necessitating some delayed reference resolving. This could also be done by adding one more pass to the conversion; however we do not believe the complexity would be much different.

Along with associations also multiplicities and different kinds of aggregation (aggregation or composition) need to be considered. Coral does not seem to have full support for multiplicities, so sometimes they are slightly modified. Support for aggregation and composition is effectively a matter of mapping a TAU enumeration to a Coral enumeration.

4.3.7 State machines

The really tricky objects in this conversion are state machines. While states are represented in TAU in a fairly simple way, the transition objects do not have a source attribute, but the source information is encapsulated in separate objects called *MultiState*s.

Much of the complexity in the conversion of transitions comes from the fact that they are quite differently represented in TAU and in Coral UML 1.4. For example, in TAU the destination of a transition is embedded in the action as a *NextStateAction*, which may or may not be inside one or more *CompoundAction*s. This means the target of a transition is not plainly visible, but is buried in the action of the transition. Hence some digging has to be done to retrieve them, and the NextStateAction has to be ignored (and usually some CompoundActions flattened) when parsing the action.

Theoretically in the TAU "u2" format it could also happen that the NextStateAction would be conditional, i.e. inside an "if" or in one of the branches of a DecisionNode. However we have yet to see an example of such a TAU model.

As an additional point of complexity, state machines can be nested as submachines in other state machines. In that case they can be statically embedded in the state machine as composite states, as the SMUML toolchain currently does not support submachine states. In nested state machines there are *ReturnAction*s. Also the transition handling needs to contain all

the complexity produced by handling decision nodes, which are represented as *DecisionActions*. These are converted into UML 1.4 choice pseudostates.

Triggers and guards also need to be implemented. The triggers need to be associated with the signals themselves, not just textually represented. Guards in TAU are expressed using the TAU action language and need to be converted to corresponding Jumbala expressions.

4.3.8 Action language

The TAU action language is syntactically different from Jumbala. The conversion should probably be rather mechanic, also implemented as tree-mapping, and not paying too much attention to the beauty of the resulting Jumbala code. We have not yet examined the TAU action language in detail, but it is to be expected that some constructs exist that are difficult to express in Jumbala. This is also important in that sending signals is done using the action language.

The CompoundAction flattening can be done by discarding the actions that are not needed (i.e. the NextStateAction) and after that embedding the rest of the actions in a single string of Jumbala code.

5 Implementation

We have implemented the first part of the conversion, the TAU to Coral TAU model conversion. Also for this we have implemented the metamodel conversion.

Both of these conversions are implemented as Python scripts. To facilitate the conversion, the Coral TAU metamodel needs to be generated first from the TAU metamodel using the metamodel conversion script. After this a TAU model can be converted into a Coral TAU model by running the TAU model to Coral TAU model converter script. This produces an intermediate file in the Coral TAU format that can be fed to the future Coral TAU to UML 1.4 converter.

5.1 Metamodel conversion

State machines, states (including composite states), transitions, actions (including decision actions), triggers, guards, expressions and signals were implemented at this point in the hand-made Coral TAU metamodel. This was done prior to discovering that the TAU metamodel is available in the TAU SDK, when it still was our target to implement a large enough subset of the TAU language in this first stage Coral TAU metamodel to support some reasonable TAU models.

The conversion from the TAU metamodel model ("u2" files) to a Coral model conforming to the hand-made Coral TAU metamodel is done using

a script named `u2tocoral.py`. It takes as its parameters the names of the input files (there may be several, as even the metamodel provided in the TAU SDK spans several files), and as the last argument the name of the output XMI file to generate.

After this a simple line-based representation of the Coral TAU metamodel is generated from the Coral XMI model of the TAU metamodel using a script named `coraltau2nat.py`. The format of the representation generated is the input format of a script named `natural2smd.py` provided to us by the authors of Coral. This script in turn generates an SMD format metamodel understood by Coral [11].

5.2 Format conversion (Telelogic TAU to Coral TAU)

The first part of the conversion chain, i.e. the conversion from TAU models into Coral models corresponding to the Coral TAU metamodel, is a Python script that reads the TAU model into a tree using Python XML libraries and traverses the tree, creating corresponding Coral objects on the fly. This conversion is fairly simple, although there are some corner cases that need to be taken into account. Identifier expressions are resolved, except in cases where it is hard-coded into the script that there really is an identifier expression in that place in the metamodel. This information could also be extracted from the converted metamodel itself, but as the cases are relatively few, it was determined that hard-coding the information is more appropriate. The script also recognizes *Integer* as a predefined type; likewise, binary operators like "==", "=" and "+" are predefined and hard-coded into the script.

Another source of extra complexity in the first phase script is the slightly inconsistent capitalization of element names in TAU. Normally attribute names are lowercased in TAU, as in *root*, *operation* or *port*. These are represented in the u2 file as "cRoot", "cOperation" and "cPort". The script removes the initial letter (c in most cases, r in others) and changes the second letter into lowercase. There are some exceptions to the rule that attribute names are lowercased, however. These are *Type*, *IsElseAnswer* and *IsInformal*. These cases were hard-coded into the script as exceptions that cause the first letter not to be lowercased.

In summary, the format conversion is relatively simple in principle, but has a lot of corner cases that need to be taken into account, making the script more complex.

6 Analysis

We think the design we have is quite capable of handling the TAU to Coral TAU model conversion of even complex TAU models. Dividing the conversion into two phases gives the advantage of being able to do the more complex part, the Coral TAU to UML 1.4 conversion, entirely within the

Coral framework. The latter phase of the conversion is planned roughly and outlined for some of the simpler features of the language.

We also have implemented the first phase of the conversion and are satisfied that it can handle complex TAU models – indeed even the metamodel itself – and that possible remaining corner cases are a matter of adding more simple handlers for the peculiarities of the TAU format and language.

6.1 Related work

To our knowledge, no previous work has been done on converting TAU models into Coral models.

A lot of this work depends on Coral [1]. Coral is a "Metamodel kernel for transformation engines". In this work, Coral's format and Jumbala is used as the input language of the SMUML toolchain, and Coral's Python API is used to read the input files.

In [5], a somewhat related problem was tackled. In it a conversion from Rational Rose RealTime UML to XMI UML was implemented.

Another approach to model transformations is to use a rewriting engine. One such approach is documented in [13]. UML models can be represented as graphs, and a generic graph rewriting engine can with a little customization be used to rewrite UML models. An interesting example closely related to what we did provided in [13] is a relatively simple rule (for a proprietary model rewriting engine described in [10]), for flattening UML state machine hierarchy. The rule example provided consists of 16 lines of simple C-like syntax.

Using a rewriting engine is a much heavier approach, useful only when a large number of transformations need to be supported or when there is need to be able to rapidly develop new transformations. The efficient transformation engine described in [10] consists of approximately 12,000 lines of mainly C++ code, not including test cases.

7 Conclusions

A toolchain for converting Telelogic TAU models into Coral models corresponding to a converted TAU metamodel has been developed, and the conversion from this to Coral UML 1.4 models has been designed and outlined. With this toolchain, after some more development, it should be possible to design the UML test models of the SMUML project using TAU.

The most important features still missing from the design are ports, which need a fair amount of work because UML 1.4 does not support ports so they have to be converted to something equivalent, and the TAU action language. Work on both is planned in the near future.

In future work it might be beneficial to try to use the TAU API instead of parsing the files created by TAU. One would think that the API would be

potentially more stable than the file format.

7.1 Acknowledgements

We wish to express our gratitude to the following parties for making this work possible by funding it:

- Finnish Funding Agency for Technology and Innovation (Tekes)
- Nokia Oyj
- Conformiq Oy
- Mipro Oy

References

[1] Marcus Alanen and Ivan Porres. Coral: A metamodel kernel for transformation engines. In D. H. Akerhurst, editor, *Proceedings of the Second European Workshop on Model Driven Architecture (MDA)*, number 17-04 in Tech. Report, Computing Laboratory, Univ. of Kent (2004), pages 165–170, Canterbury, Kent CT2 7NF, United Kingdom, Sep 2004. University of Kent.

[2] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, 2nd Edition.* Addison-Wesley, 2005.

[3] Jori Dubrovin. SMUML: Supporting the Tau action language. SMUML Project Deliverable, March 2006.

[4] Jori Dubrovin. Jumbala — an action language for UML state machines. Research Report A101, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, March 2006.

[5] Shankar Gopinath. Real-time UML to XMI conversion. Master's thesis, Royal Institute of Technology – Computer Science and Communication, Stockholm, Sweden, 2006.

[6] Object Management Group. *OMG Unified Modeling Language Specification, version 1.4.* 2001.

[7] T. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM, 50(1)*, pages 63–69.

[8] Jukka Honkola, Sari Leppänen, and Teemu Tynjälä. Modeling the SpaceWire architecture with Lyra. In *ACSD*, pages 15–24. IEEE Computer Society, 2005.

[9] Ilkka Niemelä. Symbolic methods for uml behavioural diagrams (SMUML) (project plan). March 2006.

[10] Kimmo Nupponen. The design and implementation of a graph rewrite engine for model transformations. Master's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory of Information Processing Science, 2005. Available as of writing from `http://www.niksula.hut.fi/~knuppone/thesis.pdf`.

[11] Ivan Porres. A toolkit for model manipulation. *Springer International Journal on Software and Systems Modeling*, 2(4), 2003.

[12] Telelogic. Telelogic TAU 2.7. Software. `http://www.telelogic.com/`.

[13] Tommi Vainikainen. Applying graph rewriting to model transformations. Master's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, 2005.